MOBILE APP DEV

Our Preferred Android Development Tools and Patterns

November 19, 2019 | By: Chee Yi Ong

At Punch Through, we work on both greenfield and legacy app projects for our clients. For greenfield Android projects, we generally try to conform to a modern, well-documented, and well-supported architectural pattern that's easy to follow. The Android development landscape is an ever-changing one, and up until recently, Google's sample codes on their developer portal had remained quite un-opinionated and didn't provide much direction to developers who were just starting out.

In May 2018, all this changed as Google released their Jetpack library, "a set of components, tools, and guidance to make great Android apps." It became clear that the folks on the Android team do have a few ideas on what they think should be the norm for Android development — for instance, the single-Activity MVVM pattern is the preferred paradigm for new Android apps. Now, there's even a full-fledged guide to app architecture, which is very refreshing to see including many Android app development tools.

We took a hard look at all the different tools and patterns available to us and decided to adopt new best practices championed by Google in their guide, and then some. Consider this the Punch Through Android team's starter pack for new Android apps.

Programming language – Kotlin

We started exploring and using Kotlin in our projects last year, and have found it to be fun and more pleasant to read, much like how Swift is generally nicer to work with than Objective-C for iOS. With Kotlin recently becoming Google's preferred language for Android apps, we strongly feel that Kotlin should be your go-to language if you're starting a new project.

That said, we do think there's still value in learning Java, simply because both Android and Kotlin trace their roots back to Java. Besides, there are still a ton of libraries and example code out there that were written in Java. Being able to understand them as well as you understand code written in Kotlin is an extremely useful skill to have as an Android developer. If you're familiar with Java or even Swift, we think the official basic syntax and idioms documentations are very helpful in getting folks up to speed with Kotlin and its language features.

App architecture – MVVM + LiveData

Model-view-viewmodel (MVVM) is a well-known architecture in both iOS and Android development. Notably, most of the earlier examples of the MVVM pattern utilized RxJava, which is easy to learn but requires a significant amount of effort and time to master. The release of the Jetpack library introduced the architectural component. It also provided great solutions like ViewModel and LiveData that allow us to have an observable/reactive paradigm in our projects without having to use RxJava, and that's when MVVM really took off on Android for us.

MVVM primarily introduces the concept of a ViewModel with the goal of separating business logic from Activity classes. In the MVVM world, Activity classes are only responsible for updating the UI and making some OS API calls — everything else resides in the ViewModel classes. An additional feature of ViewModels is that unlike Activities, they survive configuration changes, thus making ViewModels the optimal place to perform network calls and BLE operations, as well as storing transient states that can be used to reconfigure the Activity UI after an Activity restart due to configuration changes.

LiveData is a lifecycle-aware observable wrapper around objects, meaning we don't need to keep track of something like RxJava's Disposable and having to call `dispose()` manually, which is prone to programmer error. By having LiveData wrap our UI states as ViewModel properties, and having our Activity observe changes to the LiveDatas in its `onCreate()`, our Activity can react to changes in the LiveDatas' values and also reconfigure itself in the event of configuration changes.

Networking stack – Retrofit + Moshi converter + Coroutines

Performing network calls is a functionality that is core to a lot of apps, and Google's developer training page recommends using Volley to do so. However, we recently swapped out Volley in favor of Retrofit as LightBlue's networking library. Retrofit requires less code to be written, has more frequent updates, and garnered a more active community.

Using Retrofit feels like magic and is as simple as defining an interface describing the network call. This interface specifies the URL path relative to the endpoint, the HTTP method for the request, and any additional header fields you'd like to include.

Here's an example showing the interface for a Mailchimp API call that our LightBlue® app performs when a user signs up for our newsletter:

```
interface MailChimpService {
        @Headers(MAILCHIMP_AUTH_HEADER)
        @POST ("lists/{listID}/member")
        suspend fun registerUserEmail(
                @Path("listID") listID: String = MAILCHIMP_LIST_ID,
                @Body request: RegisterUserEmailRequest
        ): Response<ResponseBody>
}
```

Another *huge* plus is that as of version 2.6.0, Retrofit added official support for Kotlin coroutines, which is a way of writing idiomatic, nonblocking, and potentially asynchronous code that is supported natively by Kotlin. By making our function in the API definition interface a suspending function, we can then write our network code in a ViewModel like so, without having to worry about setting up callback interfaces:

```
val response = mailChimpService.registerUserEmail(
        request = RegisterUserEmailRequest(emailAddress)
)

if (response. isSuccessful) {
        // Handle success case
} else {
        response.errorBody()?.let {
                // Handle error case
        }
}
```

You may have noticed in our code snippets above that there is a RegisterUserEmailRequest that encapsulates the HTTP body for the request. Retrofit converters make it possible to define a class that you can use to describe a request body, and there are a few of these to choose from, but we opted for the Moshi converter simply. Moshi is a JSON library written by Square — the same folks behind Retrofit — and it explicitly supports Kotlin.

Here's what our RegisterUserEmailRequest class looks like:

```
@JsonClass(generateAdapter = true)
data class RegisterUserEmailRequest(
        @Json(name = "email_address") val emailAddress: String
        @Json(name = "status") val status: String = ""
)
```

Advanced dependency injection – Dagger 2

A word of caution: We don't use nor do we recommend Dagger 2 for *every* project we work on. It can be hard for newcomers to understand very quickly what it is and how to use it. But if your app is sufficiently complex and you've found dependency injection to be a recurring pain point, Dagger 2 *might* be for you.

Dagger 2 comes with a steep learning curve, from setting up your modules and components to tying it all together in your classes that actually consume these dependencies>But once you have it all set up, you'll be able to achieve true separation of concerns between producing an object and consuming it.

For our LightBlue® app, our team has used it with great success in enforcing the singleton pattern using the @Singleton annotation —and for getting access to things like our SharedPreferences wrapper class, our Firebase analytics wrapper class, and our Retrofit instance, all with the consuming party not having to know about how the dependency is constructed or where the dependency is coming from.

If you do decide to go ahead and use Dagger 2, here is an official and incredibly informative guide on optimizing Dagger 2 usage with Kotlin that we think you should read.

Coming back full circle with the example code snippets we provided above for Retrofit, here is an example of how our MailChimpService interface is constructed and provided to consuming classes:

```
@JvmStatic
@Provides
@Singleton
fun provideMailChimpService(): MailChimpService = Retrofit.Builder()
        .baseUrl(MAILCHIMP_BASE_URL)
        .addConverterFactory(MoshiConverterFactory.create())
        .build()
        .create(MailChimpService::class.java)
```

Most importantly, go with what you're comfortable with

Despite Google showing their cards and announcing their preferences, there is no one way to go about starting a new Android project. The project's needs ultimately drive its architecture, and every project has different needs. It's also likely that these "best practices" will change in the near future, so we recommend that you go with what you're comfortable with and decide which Android app development tools work best for you. For example, Java isn't going away anytime soon. If learning Kotlin doesn't feel like a worthwhile time investment, then by all means stick with Java!

As we alluded to in a few spots earlier, these Android app development tools have also been tested in multiple projects now, and they also power LightBlue® on the Play Store. We hope this is helpful for anyone just starting out or who's curious to learn about the modern ways of developing Android apps.

Stay tuned for more Android tips and tricks articles!

Chee Yi Ong

Mobile Software Engineer at Punch Through. He specializes in Bluetooth-connected iOS and Android apps. Smart as a whip, talented writer, and teacher. He enjoys gaming, photography, coffee, and traveling.

RECENT POSTS

Meet the Team – Tyler Grunenwald
Erin Moore

Mastering all 5 Kotlin Scope Functions
Chee Yi Ong

How to Delete Bonds with the Nordic nRF5 SDK
Amanda Christiansen

Meet the Team – Matt Kruger
Emily Hinderaker

CONTACT

MINNEAPOLIS
123 N. 3rd St.
Suite 500
Minneapolis, MN
55401

info@punchthrough.com